Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Marenco Turi Gualtiero Discussed with: S.Perpeli, V.Perozzi, V. Immonen, M. Sobolev

Solution for Project 5

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you a parallel space solution of a nonlinear PDE using MPI.

1. Task 1 - Initialize and finalize MPI [5 Points]

2. Task 2 - Create a Cartesian topology [10 Points]

A 2D domain decomposition strategy was designed and implemented to partition a square computational grid across an arbitrary number of MPI processes. This method supports any grid size and process count. `MPI_Dims_create` is used to determine an optimal decomposition of processes into two dimensions.
A non-periodic (with no wrap-around communication) cartesian topology was created using `MPI_Cart_create`, enabling the representation of neighbors. Neighboring relationships, on which halo exchange communication is based, were established with `MPI_Cart_shift`. The use of cartesians coordinates, retrieved via `MPI_Cart_coords`, provide the mapping of processes to their respective sub-domains . Each sub-domain's bounding box and internal data points are computed based on its Cartesian rank. This guarantees load balancing by evenly distributing grid points across processes, as sub-domain sizes are derived from the global dimensions and process count. This minimizes communication overhead by ensuring that only neighboring processes exchange data, reducing inter-process communication costs compared to more fancy partitioning schemes.

3. Task 3 - Change linear algebra functions [5 Points]

To parallelize the linear algebra kernels, the *hpc_dot* and *hpc_norm2* functions were modified to utilize `MPI_Allreduce` for aggregating local computations across processes. Other vector to vector operations and initialization routines, such as hpc_fill and hpc_axpy, were not modified as they dont require interprocess optimization.

4. Task 4 - Exchange ghost cells [45 Points]

To implement ghost cell exchange, `MPI_Isendrecv` was used for non-blocking point-to-point communication to transfer boundary data between neighboring processes. Ghost cell values were copied into the respective send buffers and received into boundary buffers as requested. Communication requests were tracked with `MPI_Request arrays`, and `MPI_Waitall` was called to ensure completion before boundary computations.

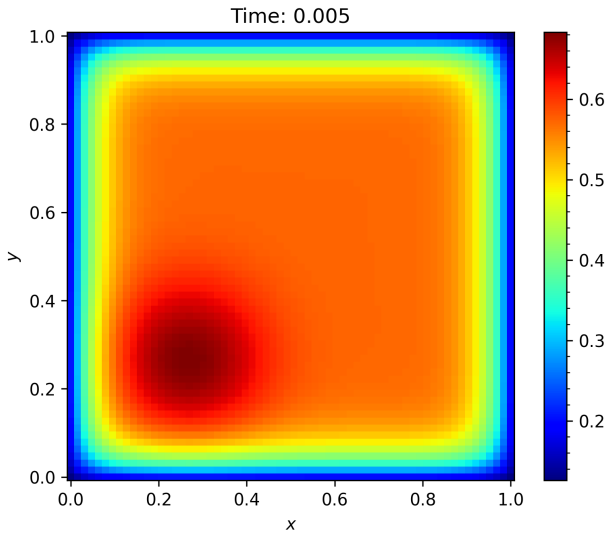


Figure 1: Correct comunication achived

4.1. Strong Scaling

In order to collect the results for the strong scaling test a sbatch file was created.

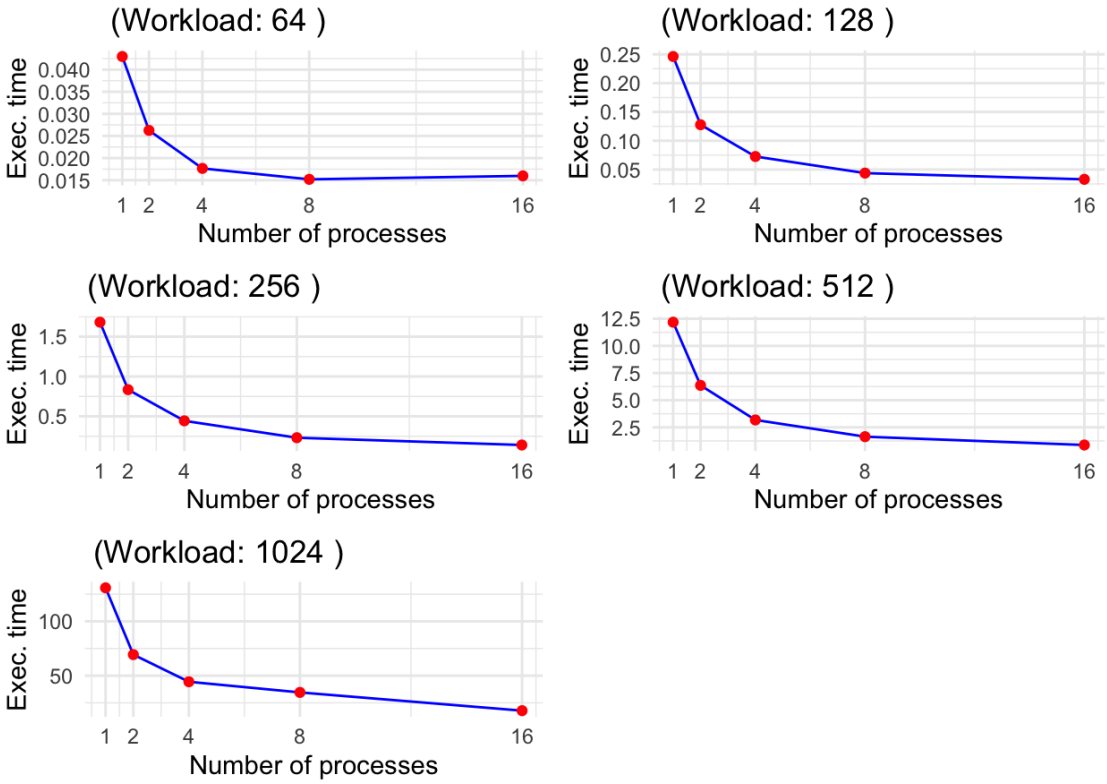


Figure 2: Strong Scaling Test for MPI

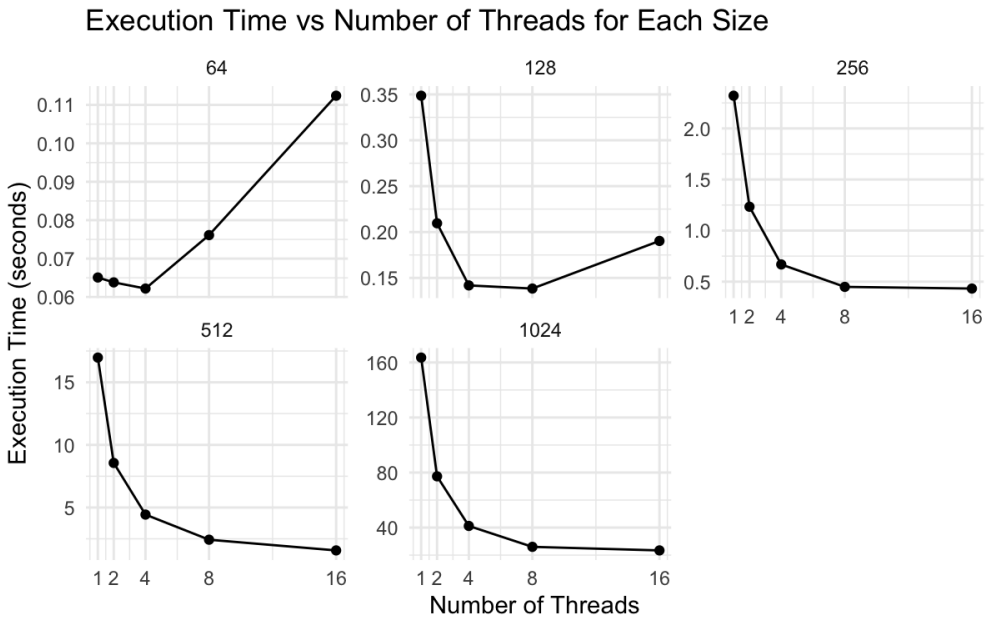


Figure 3: Strong scaling results from Assignment 3

Both OpenMP and MPI results follow the expected pattern of strong scaling, where increasing the number of processes reduces execution time. However, the efficiency of this reduction varies across workload sizes and implementations, providing insight into the impact of parallel overhead and workload distribution.
For smaller workloads, such as 64x64 and 128x128, the parallelization overhead plays an important role. In these cases, the cost of managing multiple threads communications and synchronization outweighs the computational savings gained by dividing the workload. This results in diminishing returns as the number of processes increases. This behavior aligns with the theoretical expectation that parallel efficiency declines when the workload per thread is insufficient to offset the fixed overhead costs. Moreover, this inefficiency can sometimes result in a performance degradation for higher thread counts, as observed in smaller workload cases for OpenMP, but it's mostly avoided for MPI, which also is means that MPI outperform OpenMP for small workloads and high process number.
As the problem size grows, the benefits of parallel execution become more pronounced. For intermediate workloads like 256x256 and 512x512, the larger computational demands start to dominate the runtime, mitigating the proportional impact of parallel overhead. Both OpenMP and MPI exhibit good performance improvements as the number of processes increases.
For the largest workload, 1024x1024, parallelization achieves its greatest impact. The substantial workload ensures that each process has enough computation to minimize the relative cost of communication and synchronization overhead. In this regime, the execution time decreases significantly with the number of processes for both implementations. OpenMP, however, slightly outperform MPI.
Once again, while smaller workloads are hindered by overhead, larger workloads allow for more effective utilization of parallel resources. The plot was obtained through Rstudio code which can be consulted in the "plot making" folder.

4.2. Weak Scaling

A Sbatch file has been made for this test, workloads, now depending on processes count, were pre-computed separately.
To analyze scaling with a constant work-per-thread approach, the problem sizes are set up so that as the number of threads `N_CPU` increases, the total problem size expands proportionally. This method ensures that each thread consistently handles the same amount of work, regardless of the thread count. Here's how it looks for each base resolution:

Base Resolution 64x64

- With 1 thread: starts at 64 x 64 .
- With 2 threads: problem size is 90 x 91 .
- With 4 threads: problem size is 128 x 128 .
- With 8 threads: problem size is 181 x 181 .
- With 16 threads: problem size is 256 x256 .

Workload per process: approximately 4,100

Base Resolution 128x128

- With 1 thread: starts at 128 x 128 .
- With 2 threads: problem size is 181 x 181 .
- With 4 threads: problem size is 256 x 256 .
- With 8 threads: problem size is 362 x 362 .

- With 16 threads: problem size is 512 x 512 .

Workload per process: approximately 16,400

Base Resolution 256x256

- With 1 thread: starts at 256 x 256 .
- With 2 threads: problem size is 362 x 362
- With 4 threads: problem size is 512 x 512
- With 8 threads: problem size is 724 x 724 .
- With 16 threads: problem size is 1024 x 1024 .

Workload per process: approximately 65,600

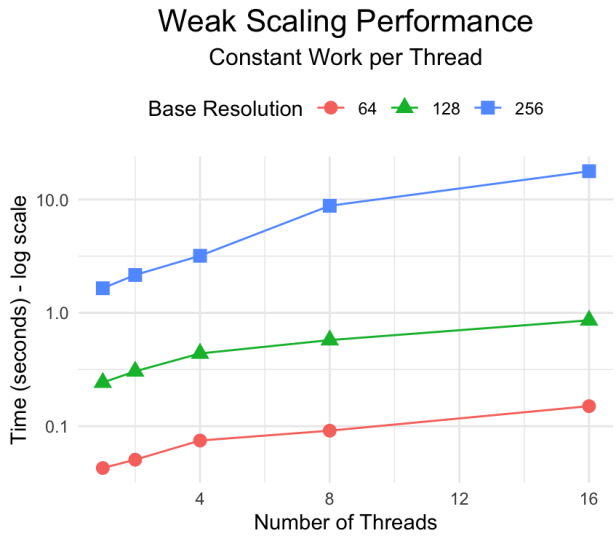


Figure 4: MPI weak scaling test

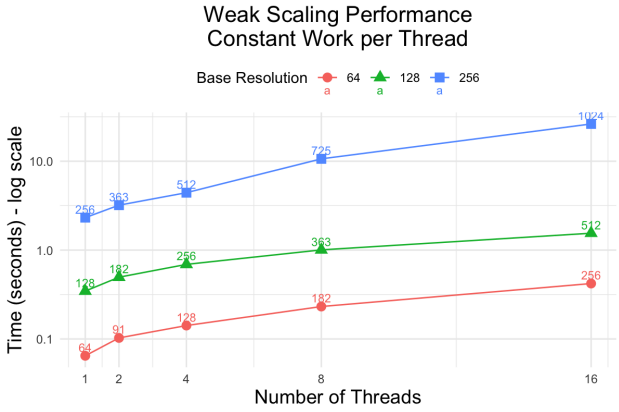


Figure 5: Assignement 3 weak scaling results

Weak scaling assumes constant work per thread/process as the problem size and computational resources increase proportionally. Ideal weak scaling yields constant solution times; deviations occur due to communication overhead or synchronization delays and memory contention.

For lower base resolutions (64,128), the MPI implementation outperforms OpenMP directly proportionally to the number of Threads/Processes. However, for larger problem sizes, both implementations balance computational and overhead costs, resulting in similar scaling efficiency. The plot was obtained through Rstudio code which can be consulted in the "plot making" folder.

5. Task 5 - Testing [20 Points]

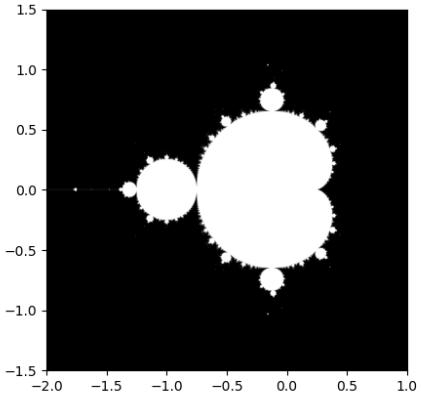
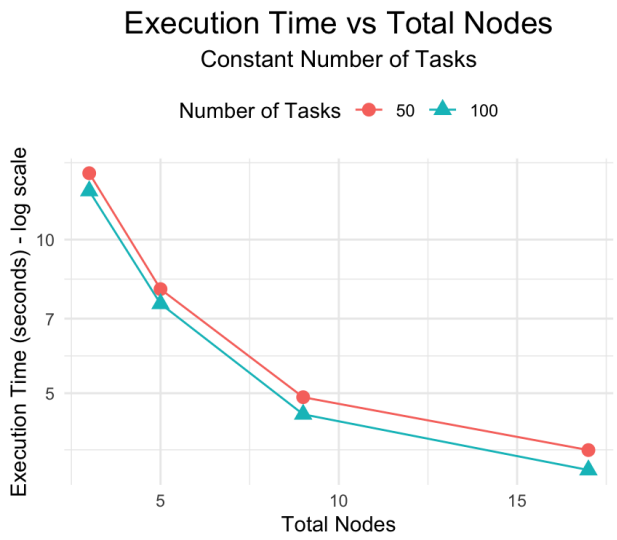


Figure 6: The resulting Mandelbrot

The implementation of ghost cell exchange is rather straightforward and the program works as expected. This time data was collected by hand by running the code multiple times with different specifics to perform the analysis for the Mandelbrot computation. The total number of nodes is the number of Workers + 1 (the Manager node).

TotNodes	NTasks	ExecTime
3	50	13.492113
5	50	7.996458
9	50	4,910835
17	50	3.868155
3	100	12.454589
5	100	7.485736
9	100	4.545795
17	100	3.535298

Table 1: Results from testing Mandelbrot's python implementation



This implementation demonstrates effective scaling as the number of nodes increases, with execution time decreasing substantially from 3 to 17 nodes. Both configurations (50 and 100 tasks) follow similar processes, showing that task parallelization impacts performance effectively.
The performance improvement from 50 to 100 tasks is due to finer granularity in the task decomposition. By splitting the Mandelbrot set into smaller patches, the algorithm better balance the workload among workers, especially in regions near the center of the set where computations are more intensive. This reduces idle time for workers and improves efficiency, particularly for larger numbers of node. Again, the existence of an overhead cost can be seen confronting the nodes as execution time doesn't decrease linearly with the increasing of nodes.